Collective Knowledge and Collective Cognition in a Software Development Team: A Case Study

A thesis submitted in
partial fulfilment of the
requirements for the
degree of

MSc New Media, Information, and Society

London School of Economics
and Political Science

Supervisor: Prof. Robin Mansell

Student examination Nr. 83252

Abstract:

This paper tackles the problems of a start-up small team software development from a collective knowledge and collective cognition perspective. The first part outlines the theory discussion as found in the recent literature, identifies some of their weaknesses and outlines an approach based on Niklas Luhmann's theory of social systems. This new approach is characterised by a clear distinction between individual and collective cognition, as it defines collective knowledge as a communications process, and thus facilitates the inclusion of computer-mediated conclusion.

A case study based on a participant observation complemented through a two weeks participant observation as a member of the team gives the evidence on which the findings are based: the coding process and automated testing are found to be processes of collective cognition not described before. This insight is facilitated only through the use of the autopoietic perspective introduced.

Contents

Introduction

A glimpse into the past

Although the great boom in the information technology sector seems to be over, it is most obvious that it will continue to invade all aspects of life, and in spite of the bust on the technology stock markets all over the world, including the spectacular decline of the dot.com phenomenon, this implies a continued growth of the significance of software development. Software development has from its very beginning nurtured its self-image as an arcane art. This attitude emerged from the roots of the IT industry in the 1960s, when technologists were accorded responsibility and autonomy. Due to the restrictions in hardware capabilities, the programmers were allowed a great deal of creativity and autonomy to find ways to overcome those shortcomings of the hardware. In the 1970s, the hardware restrictions grew less pressing, and allowed for software development to enter the focus of attention. The promises of increasing benefits caused ever-bigger systems to be attempted. It was then, that the field of software development was recognised as being overly complex, and remedies were sought in employing high-level programming languages and structured development methods that facilitated the splitting up of projects into small modules that could be coded more or less independently. The refinement of those technologies and methodologies has continued until now, and newly upcoming fashions promised time and time again to alleviate the problems of coping with the ever-growing complexity of software problems (Friedman 1993). In the beginning of the eighties modular programming languages and relational database management systems were considered state of the art, later, the attention shifted to object oriented (fourth generation) programming languages.

Problems

Yet, reports on failed software projects are abundant, and all those “great advances” do not seem to have delivered their promises. As a cause for the failure of software projects (The Standish Group 1995) identifies *requirements issues* as a primary cause for the failure of software projects. However, as (Waltz, Elam et al. 1993) have shown that even though requirements given to a software development team may be complete, they may in retrospect be regarded as incomplete, because a great part the

information conveyed from the users was forgotten. Considering the description of the software development process as overly complex, and the fact that it seems that much of the information is lost within the software development teams themselves, it seems appropriate to have a closer look on the ways in which those teams share, manipulate, integrate, store, and foremost retrieve knowledge, if one wants to gain a better understanding the source of the problems involved.

The plan of the paper

This thesis is based on an understanding of collective knowledge and collective cognition as processes. This means that these terms refer to actual social activities of knowledge sharing and knowledge manipulation – cognition – in an autopoietic relation of communications. Its goal is to identify those practices and media, which constitute this autopoietic process, and understand their interrelations and implications. In order to explore the field of software development teams in an orderly and scientific fashion, a two weeks case study has been conducted at a Hamburg-based Internet start-up called 'Tiger'[1]. It has been complemented by semi-structured interviews with all of the development team's members.

In section 1.1 and 1.2, the concepts of collective cognition and collective knowledge that guided this case study is set out. It is shown in 1.3 how an autopoietic approach can consolidate these two concepts into one which much better explains the phenomena we seek to describe, in particular utilising Husserl's concept of memory and Niklas Luhmann's theory of social systems. In 1.4 a concept of micro politics and power as proposed by Michel Crozier and Erhard Friedberg is introduced, which will help to see some aspects crucial to collective cognition, which cannot be explained by the approach proposed.

After setting out the methodology in 2.1, the methodology of the case study is introduced. In 2.2, 2.3 and 2.4 the empirical evidence provided by the case study is presented. Starting from the purposes and claimed benefits of the practises in dealt with in 2.2, they are in 2.3 contrasted by the shortcomings of the company's actual

[1] Company name and identity have been changed due to confidentiality issues.

practises. The empirical part is completed in 2.4 by presenting the company's critical self-appraisal and plans for the future.

In chapter 3,the theoretical concepts introduced are applied to the empirical findings, and it is shown that the proposed theoretic foundation is in fact an improvement compared to previous approaches.

In the conclusion 4, a summary of the results obtained is given, the limitations of the case study are stated and an outlook on open questions and options for further research is attempted.

1 Concepts

Overview

In the chapter 1 an overview of the concepts connected to of collective knowledge and collective cognition is given. The number of terms associated with those phenomena is probably almost as large as the number of publications in this field. I shall therefore introduce a first distinction, collective knowledge versus collective cognition, which will be worked out in the next two subchapters. Collective knowledge comprises all those ways in which groups, teams, or organisations store their collectively shared knowledge. The ways in which they use, maintain, and manipulate their knowledge and bring them to bear on their activities are understood as collective cognition. The third section introduces an autopoietic approach based on the theory of social systems by Niklas Luhmann and the concept of time and memory of Edmund Husserl, on which Luhmann's theory – among others – is based.

In the fourth section, the attention turns to what cannot be explained by a collective knowledge/cognition perspective. This is above all the way in which individual rationalities affect the interaction. The theory of micro politics as elaborated by Michel Crozier and Erhard Friedberg, will be outlined briefly, which is able to account for those phenomena. Although the focus of this case study is not on micro politics, it will prove helpful to have these issues in mind when turning to the analysis of the empirical data.

1.1 Collective Knowledge

This section examines how information or knowledge is stored within groups and organisations. Although there is a large number of terms being used for the description of this set of phenomena, I shall subsume them under the term of collective memory.

Storing is not enough

It is by no means sufficient to just store away information, as common sense tells you, and a brief look into the literature on collective knowledge and cognition makes completely clear. Much rather it is important that this knowledge is brought to bear upon present decisions: It is insufficient to produce archives, files or other artefacts. What is essential is that those are brought to life by complementary practices, which facilitate the retrieval and the application of this knowledge to the present decisions of the organisation. (Walsh and Ungson 1991) put it that way: “Organisational memory refers to stored information from an organisation’s history, that can be brought to bear on present decisions.“

Places of storage: Retention bins

They name a number of different ways in which this information can be stored and kept alive, which they call them „retention bins“ and of which they identify six: individuals, culture, transformations, structures, ecology, and external archives.

Individuals

The first retention bin, which they handle in a slightly superficial manner, are *individuals*. Those retain information in form of their own individual experiences. This includes not only their personal retrospection to facts and events, but also their belief structures, assumptions and values. Beyond that, they entertain personal records, files, and databases as a memory aid (Brown and Lightfoot 1999).

Culture

Culture, the second bin, embraces all those ways to think, feel, and talk about problems, which are transferred to the individual by the organisation. It seems to be evident, although not stated explicitly by Walsh and Ungson, that this refers mainly to non-formal mechanisms, like certain uses of language, narratives, sagas, the

grapevine, and shared frameworks. This knowledge is particularly effective, as it is continuously repeated and transferred throughout the entire organisation.

Transformations

As *transformations*, Walsh and Ungson define that information, that inhabits „the logic that connects an input (whether it is a raw material, a new recruit, or an insurance claim) into an output (be it a finished product, a company veteran, or an insurance payment)". In this category they subsume standard operating procedures, taylorist design of work, budgeting, and market planning, procedures, rules, and formalised systems, thus activity programs in the sense of (Hedberg, Nystrom et al. 1976), but also the socialisation of new recruits.

Structures

As a further category, *structures* are introduced. However, they equate structure with organisational roles. Although such an equation seems obvious at first sight, as both concepts condition expectable behaviour it is neither really legitimate nor helpful: While *structure* describes the formal distribution of tasks and responsibilities by means of rules and positions, *role* refers to the actual reality of those aspects as it is socially realised. (Meyer and Rowan 1977), whom Walsh and Ungson cite as evidence, are not cited quite correctly. Although they do assert that the structure of an organisation is likely to reflect rationality myths from the environment of the organisation, and thus contain knowledge about its environment, they make clear that it is rather unlikely that this fits the necessities of the inner working of the organisation. Therefore, they argue that in those cases a decoupling of formal structure and actual activities emerges, which follows a logic of reliance.

Ecology

Ecology finally is the last bin of organisational memory. Here, most of the physical artefacts of an organisation are assembled. The physical layout and equipment of the workplace reflects not only previously acquired knowledge on procedures and practises, but also information on role, position, and social status of the inhabitant, and thus helps to condition the expectation with regard to behaviour and action.

External sources

Although not exactly part of its memory, the organisation can also revert to *external sources*, in order to recover lost information of its past or gain those information for the first time. On the one hand there actors in the environment of the organisation, which nevertheless deal with the organisation, like mass media, rating agencies, or partner companies, or on the other hand former employees who are retired or have left the company.

A critical look at the “Retention bins”

It seems, that the categories, which Walsh and Ungson use, are chosen somewhat arbitrarily, and that the assignment of categories has been done with a coal shovel rather than with an archaeologist’s spatula. They seem in effect to rather have tried to demonstrate the vast variety of ways in which collective knowledge is preserved, than to provide an elaborate typology. Especially the differentiation of the last three genuine retention bins is not precisely evident. Here it is much more appropriate to emphasise the distinction between formal and informal structural properties. Already with regard to transformations, it is easy to distinguish those procedures, which are formally defined, and those, which evolve from regular practice. All the same, the distinction between role and structure most evidently makes sense. Especially for the function as memory, those distinctions make a remarkable difference. While the formalised features have a superior protection from oblivion, the informal ones are much more volatile. The informal ones however are much more effective in an organisation, as they can override formal ones which might only exist on paper. Thus it seems reasonable to subsume informal transformations and roles to the culture category, whilst formalised action programmes, rules, and positions are combined into a new category of *structure* – dropping the generic transformation category altogether. In this categorisation, the socialisation of new employees finds its place within culture, as according to the mainstream of sociological literature, it transcends transporting the official self-description of groups and turns novices into functioning members with regard to manifest as well as latent structures and practises.

The taxonomy of information

Procedural versus declarative memory

Having identified the different ways in which information is preserved, the natural next step is to ask what kinds of information are retained.

The most important distinction identified by (Moorman and Miner 1998) and (Cohen and Bacdayan 1994) is that between procedural and declarative memory. Procedural memory comprises skills, routines, and activity programmes, while declarative memory contains knowledge about facts, propositions, and events.

While declarative memory can include the contextuality of information, this is not possible with procedural memory. Information stored in routines, standard operating procedures or other activity programmes, makes it neglects to declare it’s own history or rationale. This makes it hard to estimate the consequences of changing them, or the adequacy of the knowledge preserved therein.

Decision information versus context information

Thus it is necessary to introduce a further distinction: that between decision information and context information. (Walsh and Ungson 1991) discriminate accordingly between stimulus and organisational response. Essentially, context information is not only about what *has* been decided, but also about the *why* of a decision. While functional memory cannot serve with that information, it may be complemented by cultural knowledge, which can help in preserving such context information. As the most prominent carriers of such context information, besides the grapevine, sagas, and narratives are individuals, the *loss of a central person* can result in serious information loss, as (Davies and Roche 1999) have described.

Information retrieval

The ways in which this information is retrieved are just as manifold the ways of storage. In the case of routines, roles, action programmes performed regularly, and the ecology of the workplace, this retrieval is utterly simple. In those cases it happens in an intuitive and automatic way. Other forms of knowledge, however, present greater difficulties. Here, interpretation by the members of the organisation is required. Thus we are led to ascend to the level of collective cognition.

1.2 Collective Cognition

This chapter will examine how groups and organisations organise the retrieval and interpretation of their stored knowledge.

As we have seen in the last chapter, the use of knowledge in the organisation is trivial and automatically performed just in a few cases. Much more, already the identification of the particular knowledge necessary for a decision requires a thorough discussion and interpretative action, and how this information is in turn applied during the decision process, too, as already (Daft and Weick 1984) make clear. As the knowledge present in an organisation is present not only within the individuals, it is appropriate to explore the mechanisms for its modification and use on a collective level as well.

Cognition as congruency between individuals

The understanding of collective cognition that seems most evident at first sight is to conceptualise it as congruency between individuals: The respective individuals think and feel the same, and thus achieve a common basis for convergent behaviour. Yet, it is improbable that individuals with a different background, and therefore with different interpretation schemes, should develop the same interpretations. Much more likely is that they first adapt their behaviour to the views most prevalent, and that those views also get predominantly discussed within the group, as (Janis 1982) described in his work an political decisions. In order to avoid cognitive dissonances in the long term, and to be able to perform their tasks, they will most probably negotiate shared belief structures and expectations (Cannon-Bowers, Salas et al. 1993). Conducting experiments regarding the development of those similarities over time, (Lefevesque, Wilson et al. 2001) found, that this behaviour is strongly dependent on the kind of task as well as from the interdependencies of its subtasks.
But not only the convergence between individuals, also a variety of different, even conflicting views can account for collective cognition. Individuals can mutually prompt each other to share their recollections, and even contradicting views can facilitate a more comprehensive and appropriate collective knowledge. (Walsh, Henderson et al. 1988).

Cognition as interaction

(Klimonski and Mohammed 1994) make a considerable step forward in their account of collective cognition. They pinpoint the fact, that even in spite of different views a consistent collective behaviour of a group can emerge. They emphasise that coordination becomes possible, not trough consistency or avoidance of overlap but through mechanisms of interaction.

(Larsen and Christensen 1993) follow a similar path. According to them, collective cognition is foremost based on a network of substantially different individual knowledge, out of which consistent interaction and behaviour can emerge. While this individual knowledge is understood as part of an underlying network, collective cognition takes place outside the individuals, it *is* the discussion process between the group members. They stress the importance of the fact, that only those bits information, which are available to one of the group members, *and* which he makes available to the group can become effective in the group's problem solving process. The sharing of information, then, has the same function for collective cognition as the recalling of information has for individual cognition. If the information necessary for solving a problem is available to all the group members in a similar way, collective cognition will mainly be about the manipulation and processing of the information. According to Larsen and Christensen, different kinds of social decision schemata are used in this process, depending on the situation.

Metacognition

If the information is distributed heterogeneously, it can lead to the emergence of so-called meta-cognition. This is particularly necessary, if the complexity of the task or time restrictions prevent the sharing of all the relevant data in the discussion. This meta-cognition is basically a shared understanding of who is regarded as a source for certain kinds of information. In its most advanced form, it can take the form of an "I know, that you know, that I know". Such knowledge can then also lead to a differentiation of group roles, which in turn become part of collective knowledge. Here, again the distinction of position and role becomes valuable: while team members can have the same formal position, they can assume different roles in their informal collaboration.

“Group mind”

(Weick and Roberts 1993) treat the question of collective cognition. They do research on the interaction aboard an aircraft carrier under the label “*group mind*”. While they also base their understanding on a heterogeneous distribution of information among the group members, their interest is rather in procedural, than in declarative knowledge. As a prerequisite of collective cognition, they identify a disposition to heedful interaction on an individual basis. On social level, they identify a careful and effective socialisation of the group members. Especially important is the fact, that although nobody knows all the information, and everybody only sees a small part of the situation, the internalised and frequently recapitulised heedful interaction facilitates a low level of errors.

These findings match those of (Hutchins 1991). He describes the navigation of large ships in costal waters. Here the access to all the information for single individuals is precluded by a lack of time, and the big picture is completely available to none of them. Yet through heedful interaction, which transcends formal responsibilities by informal mutual help, the complex task is mastered without noteworthy breakdowns.

1.3 Getting truly Collective – an Autopoietic Perspective

Although the approaches outlined hitherto allow a quite comprehensive insight into collective cognition, certain weaknesses cannot be denied.

Problems

Firstly, they hardly address knowledge, which is stored in computer programmes, databases and archives. However much they allow individuals access to these information carriers, by classifying them as external memory aids, or subsume them to the ecology of the workplace, all of this turns to be insufficient, when archives are shared between individuals, and when computer systems mediate communications among individuals, are used as collectively shared databases, or restructure information by themselves, and trigger communications as a result.

Secondly, the exceedingly ambiguous relation of individual versus collective knowledge and cognition is hardly less problematic. While knowledge is said to be stored in individuals, as well as in actions and communications, collective cognition takes place, according to some authors, solely in the communication process. The

delimitation, as well as the intermediation between individual cognition, collective cognition, and the artefacts remain downright blurred and ill defined.

A promising solution: Luhmann's conception

For a solution of these open questions Niklas Luhmann's theory of social systems can be employed. It offers a number of benefits: a clear separation of social and individual cognition, a concept of communication that can support the separation of the latter categories, and the potential to define memory on a social level. As this approach has been successfully applied to other fields of communication research before, so for example to the interaction of those co-present (Kieserling 1999), or an the problems of growing economic heterogeneity through globalisation in(Fuchs 2001), our chances are good, that it might prove helpful in our context, too.

The basis of Luhmann's theory is to define social systems as autopoietic communication systems. This does not mean, that they are systems *in* which communication takes place, but as a matter of fact, that those systems *consist* of communications, or put more precisely: reproduce themselves from their basic elements, which are communications. Individuals, that is: psychical or physiological systems are an environment to the system, although prerequisites of its existence. Yet, they cannot determine the social system, but the system determines by its own logic which communications can and which cannot be connected to a previous communication. This relation is similar to that of a biological cell towards it's environment: A cell is dependent on other systems in its environment, as it receives water, nutrition, light, a certain pressure and temperature from them. Nevertheless, the reproduction of cell structures, metabolism, cell division, and even the reactions of the cell towards changes in its environment follow the logic that is incorporated within the cell. Notwithstanding that the environment exerts an influence, which is vital, it can only irritate the cell or in extreme cases withdraw the preconditions for its reproduction, but it can not directly affect the structural logic of the cell.

Luhmann defines a single communication as the unit off message, information, and understanding. Consequently *understanding* in this context means *socially realised understanding*, or put differently, that the message is connected to by another

meaningful message[2]. What can be regarded as meaningful is defined by antecedent communications. Thus, Luhmann defines meaning as the difference of actuality and potentiality: the message becomes information only when put in contrast with what else could have been said. This in turn is determined by the antecedent communications. The fact that individuals can only irritate communication systems, but not determine them, arises exactly from that circumstance. Provided it intends to be understood, a psychic system can only try to affect communication by its messaging behaviour in a fashion that is adequate to the structural logic of the communication system. For collective cognition this implies, that one cannot refer to information, which has not been previously communicated, as (Larsen and Christensen 1993) already realised. By referring to it, one actualises this information, and gives it new importance: one literally *re-calls* it to communication. The same is true for information stored in routines, roles, and action scripts: By being actualised over and over again, they are recalled to communication. In using this concept Luhmann refers to the concept of retention, which (Husserl 1986) developed.

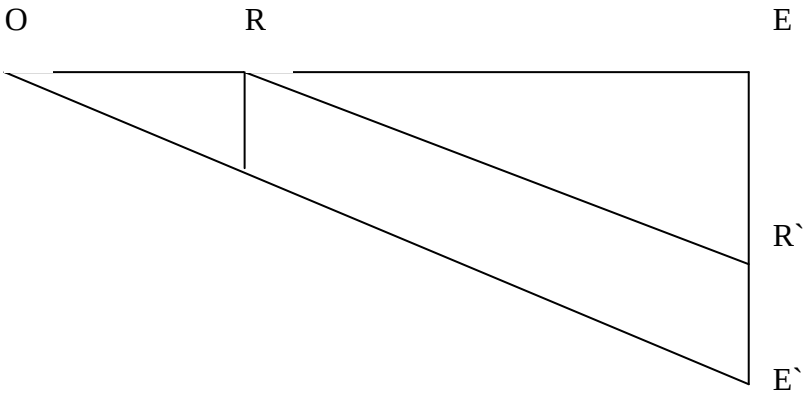
”Keeping in mind”, Husserl’s concept of “retention”

[2] A good account of the social construction of understanding, as well as a comparison and orientation of Luhmann’s systems theory compared to action theories, in particular speech act theory by Searle, J. R. (1981). Expression and Meaning. Cambridge.

is provided in Schneider, W. L. (1996). “Die Komplementarität von Sprechakttheorie und systemtheoretischer Kommunikationstheorie. Ein hermeneutischer Beitrag zur Methodologie von Theorievergleichen.” Zeitschrift für Soziologie **25**(4): 263-277.

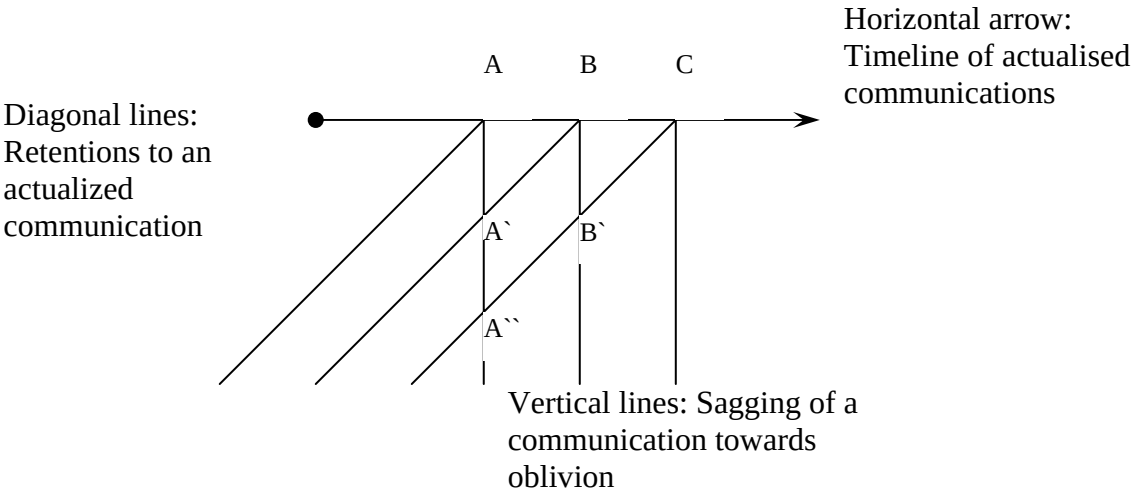
.

O represents an original communication. While time proceeds from O to E, it slowly subsurfaces, getting less relevant and giving and being obscured by the forthcoming communications, towards oblivion along OE'. The line OE represents the sequence of



following actualised communications. Whenever the original communication O gets referred to it gets reactualised, however not as O, but as retention R, which also descends along the line RR'.

A different graphical representation is proposed by (Merleau-Ponty 1974):



The sequence of communications always refers to antecedent communications, that is to retentions of antecedent communications. In this process, none of the retent communications appears as it was, but is always already actualised in a modified way. Each retention is in constant change towards another retention and thus produces a steady continuum of retentions.

When no more retentions to a certain communication come about, that is when a certain bit of information is no longer actualised, or a practise is no longer executed, it sags back into the past, until it vanishes in indiscernibleness.

One may argue that this concept based on communication fails, if the actions it deals with are not communications. But this is not a problem for our field of interest: If knowledge does not enter communication, or if it is not as Heidegger would put it, is "brought to language", it is neither collective knowledge nor collective cognition, but individual knowledge.

Through this perspective the problem posed in the beginning of this chapter, which was not quite adequately solved by the recent discussion can be dealt with: The use of information stored in archives, computer programs, and databases. Because communications, being the basic unit of communication systems, are conceptualised as the unit of message, information and understanding, the recourse to an archive, a computer output, or a database can be analysed as a message, which realises a retention to an antecedent message. Through this retention, the information present in the archive is, although via interpretation, actualised, and made available to communication from anew.

Thus we see, that collective knowledge cannot be seen without collective cognition. Moreover, they are both terms referring to one process, the communication process, by which collective or social reality is created through the autopoietic reproduction of communications.

1.4 Micropolitics – the Blind Spot

Due to its exclusive focus on collective phenomena the perspective outlined above is not able to address certain issues, which nevertheless have considerable relevance for its subject matter. Following (Crozier and Friedberg 1979) I will call them *individual rationalities*.

Rationality limited

In their work, Michel Crozier and Erhard Friedberg set out a theory of rationality of actors in organisations based upon the concept of *limited rationality* of Herbert Simon. This means, that their concept of rationality does not presuppose complete information or a persistent end to which it serves. The actor's goals and his strategies

change over time, with his perceived opportunities and his required level of satisfaction. He acts rationally only with regard to his momentary situation. In an organisation, like in a game, he has certain rules to abide by, and certain resources that enable him to improve his position during the course of the game.

Power games

Crozier and Friedberg identify a number of sources of power, which, if they are heterogeneously distributed throughout the organisation, enable the actor to negotiate with the other actors in the organisation. They define *power* quite generically as "the ability of one group or actor to limit the freedom of action of another group or actor". In organisations, this constitutes certain *"zones of uncertainty"* that an actor can experience, or complementary control for other actors. Crozier and Friedberg identify the following ones: *Information flow* – the actor controls the flow of information from or to other subsystems of the organisation. *Organisation rules* – the actor can modify or influence the organisational rules. Expertise – the actor has knowledge that is hard to replace. *Environmental Interfaces* – the actor controls contacts to the environment of the organisation. Yet, these zones of uncertainty can only become effective as a resource of power if a) another actor is dependent on them, and b) the actor controlling them is aware of his position. Within these constraints, every actor tries to develop a strategy that on the one hand defends his current status, and on the other hand increases his degrees of freedom. In this process of negotiation, he can trade concessions out of the zones of uncertainty he controls for advantages that only other actors can provide. Likewise, he can apply sanctions to other actors who act uncooperatively. The actor is in the situation where he has to continuously demonstrate what zones of uncertainty he controls to keep the others aware, and yet still limit their behaviour, so that no incidents occur, which might call for sanctions of superiors if too obtrusive.

The power to withhold information

Especially the control of information flow within the organisation, and the control of expertise can come to bear on collective cognition. In both cases, individual rationalities can come to bear on our field of interest. An actor can, in order to demonstrate his control over an uncertainty zone, feel compelled to only gradually

share his information with others. Or he may, in order to gain other advantages as part of a bargain only very selectively pass on information to certain partners.

2. Empirical Evidence

2.1. The Case Study

The enterprise studied

The case study is based on a Hamburg Internet Start-Up called Tiger[3], which undertakes to provide a service platform for the coordination of different suppliers and customers in the media industries. The area they approach can be characterised as trading informational products that can thus easily be transferred digitally. However, the heterogeneous market with numerous actors is lowly standardised, and for each transaction the products are transported by various different media spanning from sending digital media or lithographs by mail or courier to transferring them by email or ISDN. Although one customer has to send the same data simultaneously to many different partners, in tight cooperation with at least one other partner, there are many different conventions on how this is actually done, depending on the particular companies involved. As in each transaction up to four different companies or company departments are involved, a large amount of matching the different orders, data and products *manually* is involved. The business goal of the company under scrutiny is to provide a service platform on the Internet, which centralises these multilateral transactions and makes the manual matching and coordination unnecessary.

The Company has 23 employees, including four board members for HR & Finance, Sales, Marketing, and Technology. Two of them – sales and marketing – have substantial previous experience in the target industry, and are regarded as the knowledge base of the company with regard to the requirements of the market. The other two come from the software industries, where they had previously worked together in another start-up. The Engineering department is split into two departments: *Operations* is responsible for providing the telecommunications and IT

[3] Company name and identity have been changed due to confidentiality issues.

infrastructure, including running the web-servers providing the service; it consists of three members including one team lead. *Development* consists of nine engineers of which one is informally regarded as technical team-lead.

Tiger can thus be regarded as a typical Internet start-up, with the typical high emphasis on technology, especially software development, and low reliance on and low estimation of formal organisation and procedure. At the time of the study, the company had existed for one year. Financed through the first round of funding by one venture capitalist, a prototype had been developed and tested with a small group of pilot customers; the first production quality system had been developed from the experience made with the pilot system. It was about to be deployed to the live server, and the first paying customers were expected to use it in due course.

Research Methodology

The use of interviews as a technique for developing an appropriate understanding of the organisational reality is quite widely accepted as an appropriate technique. Interviews are a means to quickly and effectively gather a lot of data on how people perceive the reality they live in, and to gather a number of different perspectives on the field of interest from different interviewees. In organisational sociology however it has frequently shown, that limiting one’s perspective to this particular method can hide and bias a substantial amount of information. The most prominent problem is that of the interviewee guiding his answers by an idea of what he thinks is socially expected. He might, if there is some group pressure within his peer group, because he fears information to leak to his boss, or simply because he wants to “look good” with the interviewer, limit his account to the official company line, the version that is prevalent in his group, in order not to be regarded as a traitor, or report whatever he thinks the interviewer will expect him to say. Also situations where the interviewee deems a source of power affected, like zones of uncertainty controlled by him or certain informal bargains as described respective chapter above, might limit is readiness to supply comprehensive information

Besides being misled by intentional behaviour of the interviewees – which is perfectly rational on their behalf and just as understandable – the interviewees might omit certain aspects, either because they take them for granted and obvious, or because they are just unconscious of them. Especially in an environment like the company

researched, where the actual performance counts more than formal procedure, one runs the risk of missing a lot of practices that are relevant.

Therefore, the author has chosen not only to conduct interviews, but also to complement them by a two weeks participant observation in the development team of Tiger. The effect of doing so is threefold. In the first place you get some first-hand experience of the nature of work in the development team, which is filtered neither by preoccupation of the interviewees nor by normative scripts they adhere to. Secondly, the interviewees are quite aware of the fact, that they cannot easily “fool you”, as you can easily double-check most of what they tell you from your personal observation. Thus their behaviour will tend to offer their personal interpretations and helpful advice rather than giving “sugar-coated” accounts of the situation. Thirdly you have access to the tacit, pre-supposed, or informal practices that are likely to be missed by a survey based merely on interviews.

From the perspective of traditional ethnology approaches, the two-week observation period may seem quite short, especially if the goal is to really integrate into a team and be introduced to the relevant practices. However, although this limited time span is induced by the schedule of a Masters thesis, there were several aspects that counted in favour of the good prospects for this endeavour. First, the author had been familiar with that kind of field before, as has he has extended professional work experience as a freelancer in the field of information systems development. Secondly he had in worked with some of the tiger employees before, and therefore not only acquired some reputation as a true “techie” with part of the development team, but also had developed confidential personal relationships. This fact did also come handy as a door opener to get the support from the management in favour of the request of the researcher. In order to be accepted as a part of the professional group of techies, by the other team members, the researcher adapted to some degree the typical bonding behaviour of that group; this included the adaptation of a certain dress code like t-shirts, preferably with some technology-related application and jeans, most importantly though the avoidance of ties and anything that looked too managerial, and conversation subjects like open source, the preference of Linux versus Microsoft, and the advantages of the emacs editor above vi or the other way round. Further, the natural mode of interaction his former colleagues entertained with him, helped his

new colleagues easily accept him as one of theirs, including his participation in after-work leisure events.

In order benefit from the cooperative atmosphere also for the interviews, they were not conducted as a distinctive block at the beginning or end of the case study, but appointed during the entire research phase in an ad hoc fashion, just like occasional meetings for the discussion of other work-related issues. All the interviews were conducted as semi-structured interviews, which means that a list of topics the author wanted to address was prepared before the interview and sorted into a sequence that would allow for a conduction of the interview in a way similar to a casual conversation. However during the interview, it was attempted to maintain a casual atmosphere, so if the dialogue would take a direction not appreciated by the interview guide that was relevant for the research, the author would not intervene, but rather use the guideline as a checklist and reminder of topics to move on to, whenever he felt that a certain topic was exhausted. As soon the interviewee would change the topic, although the author had the impression that it should be dealt with in more depth, he would either inquire more deeply or take a note in order to return to that point later on.

From the above discussion it is clear, that this case study can be characterised as an overt participant observation. Thus, on his first day the researcher was officially introduced during one of the regular team meetings that take place in the morning on two fixed days of the week. The director for technology announced him as a student doing a two weeks field research in the context of his LSE Masters thesis, and left it to him to introduce the nature of his project. The researcher gave a short outline of his project, stating that he would like to be treated as just like any other collaborator, and in spite of the research nature and the short time frame of his presence, was going to try to be a productive team member. He assured that all information collected by him would be handled confidentially and be strictly anonymised when used in the scientific research process, and not be forwarded to anybody within the company, including their superiors. Then he asked the team members to allow him to make audio-recordings of the interviews, meetings, and discussions he observed under the conditions outlined above, pointing out that they be merely a memory aid for him, and adding that they could refuse to be recorded any time they wished.

In order to get an overview of the various kinds of knowledge stored and maintained in the organisation, it seemed best to try to obtain access to this knowledge in a process of participant observation.

Merging into the environment

Therefore, in coordination with (DT) and the informal team lead he was looking for a task, which on the one hand would allow the author to get in touch with every part of the project and on the other hand would also yield some practical benefit for the company. Since at that time there was no comprehensive plan of the navigational structure of the product in existence, it was agreed, that the author should work out such a map. The advantage of such an arrangement was that it included an interaction with the system from a user's point of view - as well as access to the source code, which might disclose dependencies and functionalities not directly visible from outside. Furthermore the author could interact with the developers in the mode of regular work, which would disclose the actual way of information processing. Whenever in the course of this process new aspects or questions would arise, they could be included in the interview guidelines. Thus the author employed a circular process of research, in the sense of the "grounded theory" of (Glaser and Strauss 1998), which permitted the immediate feedback of newly found aspects into the further research process.

In this (Glaser and Strauss 1998) sense the present case study should be understood as a work of theory generation. It attempts to make a first step towards a typology of the mechanisms of collective knowledge and cognition in software development teams. It should be obvious that it can claim neither completeness nor generalisability. Still, a lot will have been achieved, if one is successful in isolating some essential types active in this field.

2.2 What ought to be

In order to present the study in a more vivid and readable fashion, the author will in the following account of the empirical investigation refer to himself in the first person singular.

Requirement process

In the requirement process the requirements for function and look of the software are laid down. This is done in so called “requirements meetings” of the director of technology (DT) and the director marketing. They discuss which features are of the product are required and which are considered as desirable but optional. The result is a so-called *feature list*, containing the required and the desirable features and also estimates of costs. Also a priority list is worked out, according to importance and estimated costs, which determines the order in which the subtasks are to be tackled.

The overall result of the meeting, which combines the knowledge of the director marketing, concerning the area of his responsibility, and the information of his counterpart, the director technology concerning the technical details of the realization, is put together into an excel spreadsheet, which is turned into a html-file, generally accessible in the intranet server. The director technology completes the document by definitions of features, and paths of realization. The main emphasis here is not on technical details but on the details of the user interface.

The subtasks then are assigned to the developers, each of whom carefully checks, whether he has got all the necessary specifications. If he encounters soft spots he goes back to the director technology for clarification, who then takes care to update the feature definition.

Coding

Now the developer starts his job. He has a complete version of the development software at his disposal on his development computer. A version management system, called CVS (CVS 2001) administrates the various versions of the software in a development project. This system stores all files relevant for the project in all versions on a central server. The developer then can “check out” any version of the software with a special command, and then obtains copies of all pertinent files transferred to his local hard disk. Without disturbing the work of the others, he then is able to work out and test changes of the software under development in this “sand box”. And only when he is satisfied with his changes, he transfers the files back to the server (check in), so that the new version is accessible to his colleagues.

In this process the server keeps a record of the history of the files, so that one can find out at any moment, who is responsible for which changes and all previous versions can be restored.

The developer looks at the code at hand, realizes which changes are necessary and performs all changes independently. This is done on the basis of shared *code ownership.* That means that there is no formal unique responsibility of a developer for a certain part of code. On the contrary all developers may change any part of the code.

When the implementation is completed, the developer checks whether the changes were made work correctly, and sends the files back to the server. He informs the DT about the changes, who notes the completion of the task in the requirements document.

Pair Programming

In the case of certain difficult or very important tasks the method of pair programming is employed. Two programmers work together on one task: the one talks, telling the other what to do, the other writes the code.

The expectation is, that better code will result if “four eyes” control its generation. It is remarkable, that the employment of this practice is made to depend on the qualification of the programmers above all: “We employ pair programming only, if it really fits the situation, i.e. if a programmer expressly says: ‘Ok, I would like somebody else to take part and keep an eye on it.’ – I think, we cannot afford this all the time, but there are setups in which this really makes sense – but I would say we don’t just have average people, but really good people. They just would not like to have somebody breathing down their neck all the time. Usually they have enough experience to simply say ‘Oh no, I can do this by myself.’”(DT)

Testing

Manual Testing

For the testing of the correct completion of a task two methods could be isolated. The first is *manual testing.* The tester acts as a user, testing the program in a stepwise fashion, trying not to omit extreme situations in which errors might turn up. It is important to give also those kinds of errors a chance to occur, which are the results of

an interaction with other parts of the software: “Normally, when I do the testing, I run through complete sequences, say, from inputting an order, right down to the end. I try to include the more important special cases into this process, which I know to be dangerous. I like to know that even then everything keeps functioning.”(E7)

This manual testing is done by the developers on their development machine and also collectively in a “company wide test”, before a new version is played back to the public server. In addition the marketing division employs two students, for the special purpose of testing the system frequently.

This method, however, is known to be error prone: ”…everything is done by company-wide manual tests. With this you simply cannot proceed systematically, as we have seen: yesterday’s release had two bugs, which we have just not seen. […] Anyway, you see only what you want to see.”(E2)

For people from outside, it is said, it is very difficult to test in an effective way, because they are not able to judge the causes and places of errors: Testing must always be done by the developer: “If I do a program, I can know what can go wrong. So, if a field must not be empty, I can put in a space, and fool the program. An external tester can not know that.“(E6)

Automated Tests

For this reason one employs, what is called *automated tests,* which are supposed to simplify frequently recurring tests. Instead of performing some tedious process time and again, one simply writes a little program, which simulates this process and checks the results: “All these stupid exercises, how do I look at a web-site the right way, whether the right entries are in the list, all this can be automated. I want to spare the developers from the burden of doing such things.”(E2)

Unit Tests

Two kinds of automated tests are to be distinguished: The *unit test* and the acceptance test. The unit test concerns the level of software structure. It concerns modules and their technical structure. (E2) characterizes them strikingly as follows: “I see the main tasks of unit tests, that parts of the source code which a developer writes himself, can be checked fast and simply for correctness, that is those things which are really critical, which *do* something; not in conjunction with other objects, but just on the

stand-alone level, regarding their methods: I put *this* in, and I expect *that* to come out. Additionally, if I write the tests first [that is before coding the module], I ask myself: what is my object supposed to do anyway? So this is also a design tool."

Acceptance Tests

The acceptance test on the other hand considers the entire software from the vantage point of a user, who feeds the system via browser, makes certain entries, and checks the actions of the program for correctness. The test-tool can be configured, and is to be fed by the students with input, in accordance with the requirement specifications.

Bug-database

If a bug is discovered, it is it is stored in a generally accessible database together with all the relevant data, like version of software, status, importance, priority, state of repair and person responsible. A newly discovered bug is first assigned to the informal team lead, who then can edit the information, add commentaries and delegate further processing. The same holds for everybody ho is assigned a bug. Thus a developer, who has analysed the bug, but needs further information, may change the status to analyzed, add and edit commentaries and pass on the assignment to some other developer for further processing. The system is keeping track of the processing steps all the time, and is able to present an overview of the entire historical development of the bug.

Daily Builds

The concept of *daily builds* is closely connected to automated testing and quality control. The idea is to transfer the complete software to a test system, compile it completely and run all the automated tests on a daily or at least regular basis. This procedure is supposed to have the effect of an early warning system against problems. With Tiger the usage is somewhat different: "We don't have daily builds, but we integrate as often as possible. We don't have to overdo this. But if I see, the thing blows up every time, then I will have to do this more frequently. (DT)

2.3. What is

What has been described above is largely the process of software development as it should be officially. It was reconstructed from own observations and the interviews. Like in many enterprises, however, the reality is different.

Introduction to the project

In order to be able to work in reasonable way, I first had to establish a development environment. To begin with I was introduced to the internal server, which contains all shared documents by the team lead operations, who temporally was in charge of the computers, because the system administrator was on leave.

Since the development is done under UNIX, I first had to install Linux (a variant of UNIX) in a separate partition of the hard disk, as my I brought my own computer which uses windows. The central server contained Linux, development software and installation manuals. It turned out, however, in the course of the installations, that a large part of these documents were obsolete. At first I dutifully followed the manuals step by step and turned to the administrator only in cases of malfunctioning. But then I would browse through the documents and turn to the system administrator right away. He was able to put me on the right track quickly, and to tell me in a detailed way, which documents to skip and what steps to take.

After I had installed my system and had managed to start my local copy of the development software I turned to the informal team lead and asked for an introduction to the system (E5). He passed me on to (E1) who was supposed to have a good overview.

For the of introduction we sat down at his computer, and he introduced me to the structure and the technical features of the software: which kinds of modules exist, how they are supposed to interact, which are the conventions for naming them, and where to find them in the directory hierarchy.

Concerning the functioning of a part of the development tools, he turned me to the documentation on the intranet server – and likewise for a subset of the functioning of the program: “I think there is a file which explains quite well the respective roles. For the rest you best just try out the program. This is the best way for you acquire a

feeling for the functioning of the program, you will see how the whole thing works." (E1)

The Intranet

So I took a closer look at the intranet. It is a hierarchy of html- and other documents. At top level you could chose several divisions. In the directories of the respective divisions you could find links to the home directories of the respective persons. These were either empty, or contained personal information, or documents pertaining to work. Some important documents were accessible directly by links from the divisions, e.g. the requirements document. There was however no unified structure to be discerned, telling you where to find what. I found the document concerning the roles (the various permissions with respect to parts of the system: who is entitled to access or change what) by accident, after an extended process of just clicking back and forth. It contained a spreadsheet describing the roles in German and English and a sketchy description of a couple of them. In the rest of the fields, you just found "tbd" (to be done).

A glance at the requirements document presented an equally sobering picture. It contained a sketchy description of the features. The links, however, which should have led the user to the feature definitions, pointed nowhere.

This impression was shared by the majority of persons interviewed: "The impression I have about the intranet is not especially good. You never can be sure that an information is still valid. The spreadsheet contains some concepts, their roles and functions, but the links mostly lead nowhere. I havn't encountered even the semblance of complete list of the concepts we are using all the time." (E2)

Where to get information?

There exists however a working application. So the necessary information had to be hidden somewhere else. Where it came from, I tried to find out in the interviews. Naturally the problem was not particularly present for the long-standing collaborators, whereas the recently recruited ones found it really pressing. The sequence of steps in the process of information acquisition was always described as follows. First one tries to get the picture out of the code or other documents. Then the nearest colleague is asked, or somebody from the development division, of whom one thinks he knows the

kind of problem, and if that does not help, one approaches a colleague from one of the other divisions. “I first ask my colleagues from the engineering division, then the marketing and sales division. This is not so difficult in an enterprise of only 20 people”. (E7)

As to the contents of information sought, three different species of problems could be discerned: understanding the business field, understanding the requirements and understanding the code.

Understanding the business field

This area was felt to be especially problematic. “The business terms are the problem. They are just very specific, and he who does not come from this business branch, has his problems especially with the technical language. This mapping is just awfully hard, just understanding how all this depends? How do you have to understand the data model? You can understand what it is like, in the explicit form. But what is a production item, an order, etc.?” (E4)

The way most of the newer employees found out was that they „played“ with the program from the perspective of a user, till they had become acquainted with the way it functions. “On the one hand you just ask, on the others, you look at the pages and just try to get hold of the logic step by step, and finally you get it and think: Oh that’s the way it works. And if you’ve got thus far, the rest is almost logical.”

Thus it seems that the main source for understanding the business field is the program, which the developers are supposed to generate.

Understanding the requirements

As described above, the requirements were especially unclear. The developers saw this quite distinctly, but, in contrast to the problems of business field, they regarded this situation as an almost natural condition of their activity, and thus complained only little. Their procedure, as gleaned from the interviews, was as follows. The developers took the requirements document, together with business field knowledge, as a basis to guess the missing details. If they got stuck, they turned directly to the marketing division to obtain a more exact definition. Employing the information thus obtained, they programmed the module in question. The information, however, thus obtained was not put on record. „We maybe sit down and have a conversation with business

people, but then we have to take our interpretation of that and make it into workable pages, code etc.“(E1)

Thus the code seems to be the only repository of the information obtained in the process.

Understanding the code

Since the code seems to be the only possible source of detailed and general information, we now turn to this field.

The first and most obvious question put to the developers was: how do they acquire an understanding of the code. Three problematic areas could be discerned, which, however, were not seen by the developers as clearly disparate. What I got at first, were contradictory answers to my question, how to best understand code. The answers mentioned the *overarching structure* of the code first, then *what* it does, and finally *why* the code does it.

The overarching structure

Regarding the overarching structure, i.e. the partitioning of the code into different modules and files, understanding it seems to be rather simple to the developers. What is important to them is an entry point, which makes the general structure visible in an exemplary fashion, such that they can get a general picture of the system: “In any case you need the prominent entry points, they have to be formulated some way, as a web page, so that one can start from there and say: ok, what is the thing using, and how are these connected. That I can see from the code. But I need the information where to start, how get into it.”(E2)

Why and How

The last two points, concerning the “why” and “how” of the functioning of the code, were mentioned by the developers mainly in connection with the *comments*, especially marked parts of the code, which are not evaluated, but reserved for text, which can serve as commentary to the code. Whereas some said they considered comments as superfluous: “The source code itself is the best description of what it does”(E5), others remarked they knew of nothing worse than reading uncommented code. This apparent contradiction quickly evaporates, if one enquires in depth what is meant by “comment”. The enemies of commenting see it as a repetition in natural

language of the machine-readable commands. The “friends” in contrast, expect from the comments explanations of the context of coding, which say e.g., why a function was put into code this way and not differently. Also the “enemies” deem a transmission of context information necessary. „And there’s enough talking between one person and the other ‚oh, you wrote this, how does it work?’ and that everybody already knows how the others are doing.”(E5)

Interestingly, there are three persons who are generally considered to have the most relevant background knowledge: “Steve hat das system ja gestaltet, und wenn es eine frage gibt, steht er dem entwicjkler zur seite und hilft ihm, eine gute entschedung zu finden.“(E2)

As mentioned above, the developers presuppose, that different parts of the system have the same structure. This is achieved by two mechanisms: *copying* and *refactoring*.

Copying

The first kernel system was written by a small team of three people, who took care to keep the systems architecture consistent and perspicuous. Those who came later were assigned minor tasks by way of introduction. They took the existing modules as examples, or copied and modified them. Thus the structures were reproduced. „There’s enough code written right now, that everybody who writes something is either gonna modify something that exists and get a feeling for the code structure, or they are copying something that exists like doing a new PO – oh you can look at existing POs and get a feeling for it.”

Refactoring

The other method is *refactoring.* (E2) and (E5) regularly read portions of code without following a concrete aim. If they see that the structure risks to become unclear, they write guidelines for restructuring the code and have the existing code reworked “If (E2) and (E5) look into the code somewhere and see, that things become too voluminous then this is put on the table […](E2) and (E5) write the guidelines, which we put to the others to morrow. One of us or both will sit down with the others and explain to them, how we would like to proceed in a consistent way from now on”

Quality assurance revisited

In the course of my work towards the navigational map for the system, I noted two bugs. When I had fixed them, following the principle of "shared ownership" I asked my colleagues how I could start the automated tests to make sure my fixes would not affect dependent code. Silence was the answer. I had to look at faces with an irritated expression also in the neighbouring room, when I asked there. Then, after a brief period of deliberation, I was told not to worry - if my code compiled without problems. As I gleaned from further interviews, there *were* no automated test procedures – contradicting utterances of DT notwithstanding. Therefore a further means of quality assurance could not work as supposed: the daily builds.

In the absence of functioning unit tests one could only be sure of one thing: syntactical, i.e. lexical and grammatical correctness of the code.

The third tool for quality assurance was indeed found in good shape, functional and well in use: the bug-database.

2.4 The Future

The deficiencies of the actual process of development compared to the self imposed standards look shocking at first. With Tiger the people at least in part aware of the problems. They are facing them and have plans for decisive improvements. Besides there are good and well reflected reasons for the deficiencies.

First there is the time pressure, to which the company is subject, being a start-up financed by venture capital. You have to be able to show first results fast, in order to convince investors and potential customers. Therefore the development cannot wait for carefully worked specifications. „Actually the lack of specs costs most of the problems, as in any start-up company, that, and poor product management. Because you try to develop as fast as they come up with ideas."(E2) In addition the position of a requirements manager has been vacant for some time. "If three directors spend their time in a meeting in order to deliberate what feature XY should look like […] this is not our main task."(DT)

As regards automatic tests, technical reasons are mentioned, the solution of which had to wait, because of time constraints, and the higher priority of other tasks. There is also an endeavour for improvement on this front: "… what also is caused by the

paramount interest to get version 1.0 ready and out as soon as possible. We have the definite feeling that we have some homework to do if we go on to versions 1.5 and 2. We cannot go on without making sure that what we had before still works."(DT)

A newly appointed developer (E2) has started to take care exclusively of these tasks. He will be shielded from the daily business and be responsible for the development of a framework for unit- and acceptance tests, as well as for the introduction of a new groupware system, which will replace the old and opaque intranet, and which is supposed to drastically improve the sharing of information in the company. "I am looking forward to a new intranet system, which has left behind those lengthy publication processes. I hope for an new improved quality of information.[…] So I hope that each one who has something to say, puts it into the new system.[…] [In the present intranet] simply the whole hyper-link structure is lacking. You find only stand alone documents, which are not linked. If you want to find something there – even if it is there – you just can forget it"

3. Analysis

In the following our practical findings find their place in our theoretical framework. The emphasis is on aspects, which have not been treated hitherto in this fashion. It will become evident that the operationalisation of collective cognition and collective memory as autopoietic system of communications makes good sense if one wants to describe the cognitive processes mediated by the code.

3.1 Software as Collective Cognition

Thinking in Code

We have already analyzed the process of collective cognition as system of connecting communicative processes utilizing Luhmann's system theory. The elementary unit of a communication system, the communication, built up from message, information and understanding, is to be is conceived as connection of one communication to another one, where the preceding communication has a double role: it enables the succeeding communication to explicate information, because it determines the space of possibilities, in contrast to which its sense is to be explicated, and at the same time, it restricts the range of communications, which can be connected. In the process of

communication a previous message may be recalled, i.e. a retention may be formed, which “remembers” the old information.

If you want to interpret code in this fashion, you have to understand it as a message. This is quite natural. You just have to remember that programming code is nothing but a sequence of commands, expressed in a certain formal language.

If you agree to start from this obvious premise, the characteristics of code development fall into place: In what we have called the copying process - this seems to be the dominant mode in later phases of program development - the new modules are obtained from the old ones. In this process the old modules on the one hand restrict the messaging behaviour of - i.e. what can be said by - the new modules. On the other hand, they enable them to be understood as difference. In addition the modules contain function calls to other modules: they can take recourse to other messages i.e. remember them as *retentions*.

The program recalls business logic

As we have seen, the program is not only a product but also, and more importantly, a carrier of business information. The importance of this fact for the organization can hardly be be overestimated, since said information is accessible nowhere else (in a likewise concentrated form). It is also important in this context to conceive the action of the program as a message. You have to remember that *message* in the sense of communication sciences is to be understood as messaging *behaviour*. Now this behavioural feature can hardly be denied to modern computer programs. They prompt an input of certain data, confirm that a certain action has been performed, a file has been saved, etc. Here the conditioning of connecting messages is still more evident than in the case of code production. You cannot fail to notice, that in this process knowledge is remembered, made present, in a communicative way and that these communications feed back to the process of code production. I.e. the process conditions and restricts the code, which can be generated.

Automated testing as Collective knowledge

Unfortunately *automated testing* did surface in this case study only as a concept, not as a reality. Still it seems to be a method of quality assurance, generally recognized in the field. It seems to belong to the standard terminology employed by the engineers. Furthermore it belongs to the recognised practices of extreme programming, a well

known, and widely discussed methodology of development (http://www.xprogramming.com/ 2001). Therefore it seems a good idea to enter into a brief theoretical discussion of it.

For automated tests holds, what has been said about collective memory in the section on code. They contain *information* on *which* bugs are thought to be possible. The tests are special in a certain way, however. The information is recalled without the intervention of individual human cognition. The individual may start the automated test, but even that is not necessary, the *daily builds* can be automated. In this case the entire knowledge is recalled automatically. Only if certain error situations occur, an error message is output and a communication with an individual is started. However the entire body of information or knowledge, which is coded into the unit test, is recalled automatically. This way we see communicative cognition take place without human intervention.

3.3 Context knowledge as source of power – how to become a team lead

As set out in the theory part, certain features of collective knowledge cannot be explained from within the autopoietic perspective. Individual rationalities are likely to affect collective cognition especially via modification of the readiness of individuals to share information. It has been shown, that procedural memory, or decision information is available in plenty. The opposite however is true for declarative memory, and especially for context or stimulus information. There is striking scarcity of any recorded forms of this kind of knowledge. However there is a perceived primary source for this kind of information: The engineers (E1) and (E5), who have been on the project for quite a long time.

This might provide an alternative explanation to the unequal distribution of the different kinds of knowledge. Through the lack of contextual information throughout the team, they are constantly being asked to provide this to the others, which is a steady demonstration of the zones of uncertainty they control for the entire company. There is no danger of any escalation of the situation, as long as they are reasonably responsive to requests brought before them. Yet they run no risk of endangering their zone of uncertainty, as they dispense their knowledge only on an oral basis or in small portions, which it is about to be forgotten in due course. Their formal role as

developer team members does not require them to record any of that information in a formalised way, so there is no danger from that side as well. Yet, the role-shaping effect of specialised information is evident, as (E5) is already regarded as informal team lead.

However, the empirical data as available does not allow to determinedly draw such a conclusion. A separate study would have to be conducted with a more subtle methodology for data recoding and analysis, like ethnomethodology.

4. Conclusion

The understanding of the processes in software development teams has hitherto been rather weak. In spite of the evidence, suggesting that the memory maintained within the software development team can in fact be crucial to team performance and goal attainment, there is only very little research done in this field. This cases study undertakes a first step in this direction. As it enters an area little is known about, it can be characterised as an explorative case study.

The theory analysis has supplied a good number of insights into how groups process information and organise their memory. Yet, two deficiencies could be identified: The present theories had problems in handling archives and computer-mediated information and in delimiting between individual and collective cognition. To cure this, a concept of collective cognition was introduced, that is entirely based on communication as an autopoietic system.

In the company researched, it seemed at first impossible to gain substantial information on the software they developed. Yet they seemed to have a software that was well functioning.

Thanks to the theoretical framework introduced, it has been possible to identify the mechanisms that facilitated the success of the development process in spite of this deficiency: the concept of the source-code as an autopoietic communication system in which the knowledge is maintained, and that of software as communicating entity that maintained the business logic that was available nowhere else in a consistent form.

Due to the character of this case study as an explorative study, and its limitation to a single case, it neither claims completeness nor generalisability. It rather wants to be understood as the start of a research project in the sense of grounded theory.

It lays a basis for further research in two directions: Firstly a series of further empirical studies could be conducted using the theoretical sampling methodology of Strauss and Glaser, in order to generate a saturated typology starting from the findings identified in this paper. Secondly, the theoretical foundation of the autopoietic concept outlined in the theory part should be elaborated and sharpened in more detail.

Finally, a follow-up study at tiger should be quite telling: The new intranet and an actual introduction of unit tests could change the realities quite dramatically.

Literature

Brown, S. D. and G. Lightfoot (1999). History 2.0: Performing the past in the context of electronic archives. Approaches to the study of Historical Conscousness, Potsdam.

Cannon-Bowers, J., E. Salas, et al. (1993). Shared Mental Models in Expert Team Decision Making. Individual and Group Decision Making: Current Issues. C. NJ. Hillsdale, NJ, Lawrence Erlbaum**:** 221-246.

Cohen, M. D. and P. Bacdayan (1994). "Organisational routines are stored as proedural memory: Evidence from a laboratory study." Organization Science **5**(4): 554-568.

Crozier, M. and E. Friedberg (1979). Macht und Organisation. Die Zwänge kollektiven Handelns. Königstein.

Daft, R. L. and K. E. Weick (1984). "Toward a Model of Organizations as Interpretation Systems." Academy of Management Review **9**(2): 284-295.

Davies, C. A. and D. Roche (1999). Emodying organisational knowledge. EGOS Colloquium, Warwick.

Friedman, A. L. (1993). The Information Technology Field: An historical analysis. Social Dimensions of Systems Engineering. Q. P. Hemel Hempstead, Ellis Horwood**:** 18-33.

Fuchs, C. (2001). Soziale Selbstorganisation im informationsgesellschaftlichen Kapitalismus. Norderstedt, BoD.

Glaser, B. G. and A. L. Strauss (1998). Grounded Theory. Bern, Huber.

Hedberg, B. L. T., P. C. Nystrom, et al. (1976). "Camping on seesaws: Prescriptions for a self-designing organisation." Administration Science Quarterly **21**(41-65).

http://www.xprogramming.com/ (2001). **2001**.http://www.xprogramming.com/

Husserl, E. (1986). Phänomenologie der Lebenswelt

Ausgewählte Texte II. Reclam. Ditzingen.

Hutchins, E. (1991). The Technology of Team Navigation. Intellectual Teamwork: Social and Technological foundations of Cooperative Work. R. Galegher, E. Kraut and C. Egido. Killsdale, NJ, Erlbaum**:** 191-220.

Janis, I. L. (1982). Groupthink: psychological studies of policy decisions and fiascoes. Boston, London, Houghton Mifflin.

Kieserling, A. (1999). Kommunikation unter Anwesenden. Studien über Interaktionssysteme. Frankfurt / Main, Suhrkamp.

Klimonski, R. and S. Mohammed (1994). "Team Mental Model: Construct or Metaphor." Journal of Management **22(**2): 403-437.

Larsen, J. R., Jr and C. Christensen (1993). "Groups as Problem-Solving Units: Toward a new Meaning of Social Cognition." British Journal of Social Psychology **32**: 5-30.

Lefevesque, L. L., J. M. Wilson, et al. (2001). "Cognitive divergence and shared mental models in software development teams." Journal of Organizational Behavior **22**: 135-144.

Merleau-Ponty, M. (1974). Phänomenologie der Wahrnehmung. Berlin, de Gruyter.

Meyer, J. and B. Rowan (1977). "Institutionalised Organisations: Formal Structures as Myth and Ceremony." American Journal of Sociology **83**: 340-363.

Moorman, C. and A. S. Miner (1998). "Organisational improvisation and organisational memory." Academy of Management Review **23(**4): 698-723.

Schneider, W. L. (1996). "Die Komplementarität von Sprechakttheorie und systemtheoretischer Kommunikationstheorie. Ein hermeneutischer Beitrag zur Methodologie von Theorievergleichen." Zeitschrift für Soziologie **25(**4): 263-277.

Searle, J. R. (1981). Expression and Meaning. Cambridge.

The Standish Group (1995). Chaos. Dennis, The Standish Group.

Walsh, J. P., C. M. Henderson, et al. (1988). "Negotiated Belief Structures and Decision Performance: An Empirical Investigation." Organizational Behaviour and Decision Processes **42**: 194-216.

Walsh, J. P. and G. R. Ungson (1991). “Organisational Memory.” Academy of Management Review **16**(1): 57-91.

Waltz, D. B., J. J. Elam, et al. (1993). “Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration.” Communications of the ACM **36**(10): 63-77.

Weick, K. E. and K. H. Roberts (1993). “Collective Mind in Organisations: Heedful Interrelating on Flight Decks.” Administration Science Quarterly **38**: 357-381.

Appendix A – Interviewees

E1 Engineer 1

E2 Engineer 2

E3 Engineer 3

E4 Engineer 4

E5 Engineer 3

E6 Engineer 6

E7 Engineer 7

Engineer 8 was not interviewed, as he was on holiday.

DT Director Technology

The interviewees' identity has been anonymised.

The citations are quoted without any transcription marks showing the pronunciation and referenced with the interviewees' codes as shown above.
Omissions or additions by the author are marked in square brackets.
Where necessary, the translation from German has been done by the author, where possible transferring idiosyncratic formulations.

The full transcripts in German are available upon request by the LSE examiners on a confidential basis to protect the identity and confidentiality of the interviewees.